A simple linear-time planar layout algorithm with a LEDA implementation

Colm Ó Dúnlaing*
Mathematics, Trinity College,
Dublin 2, Ireland.

June 4, 1998

Abstract

This note describes a simple linear-time algorithm for realising the embedding of a planar graph. The realisations are rather 'bad' in the sense that the faces' areas can vary exponentially, but they provide a visual certificate of graph planarity, at least, for small graphs. The algorithm takes as input a triangulated planar map, and computes coordinates for each node in a straight-line embedding. The paper includes a LEDA implementation of the algorithm.

1 Introduction.

An interesting and very well-studied topic of graph theory is, given a graph G, (a) to determine whether it is planar (a term which will be defined below), and (b) if so, to produce a plane embedding of G. Part (a) was settled in a satisfactory way by (for example) the Hopcroft-Tarjan planarity algorithm [7], which runs in linear time; part (b) is still a matter of intensive study; see, for example, [1,2,3, 4,8]. This paper gives a simple solution to part (b) (using various graph algorithms, including planarity, for preprocessing). The greatest merit of our solution is simplicity. The layouts produced will probably be satisfactory only with rather small graphs.

We review some definitions (see [10]). A graph G consists of a finite set of *nodes* and a set of *edges*, where each edge $\{u, v\}$ is an unordered pair of distinct nodes of G. In the adjacency list representation of G, for each node u there is given a list of all nodes v such that $\{u, v\}$ is an edge of G. The graph is *connected* if every pair of nodes is joined by a path (sequence of contiguous edges) in G, and it is *biconnected* if every pair of different edges is contained in some simple cycle of G. It is *planar* if there exists a plane embedding of G, defined as follows:

A plane embedding of G is a map which takes the nodes to points in the plane (we shall call these points *vertices*) and takes each edge $\{u, v\}$ to a simple closed curve-segment, which we shall call an *arc*, connecting the corresponding vertices. Arcs are simple, so they do not

*Supported by the EEC under Esprit BRA 7141 (ALCOM II). This is a reprint of Report ALCOM-II-429 (1994). E-mail: odunlain@maths.tcd.ie

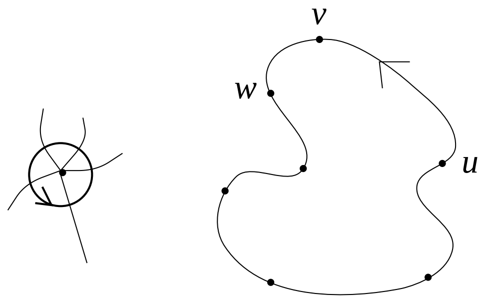

Figure 1: cyclic order around vertex and face.

self-intersect, and different arcs can intersect only at endpoints. A *straight-line embedding* is one where all the arcs are straight line-segments.

The union of all the embedded arcs and vertices forms an infinite (but compact) subset of the plane which will be called an *embedded graph*. If a planar graph G is connected then any embedding of G is connected, or more strongly path-connected, in the sense that any two points in the embedded graph can be joined by a path in the graph. The *faces* of an embedded graph are the open (path-) connected components of its complement in the plane. The *external face* is the unique unbounded face.

The arcs incident to a vertex u of the embedded graph can be ordered anticlockwise around u; thus an embedding of the graph induces a cyclic ordering of the edges incident to each vertex of the graph.

Between the concepts of planar graph and a plane embedding of a graph is that of a *planar map.* This is a (planar) graph whose adjacency lists are sorted in cyclic order compatible with anticlockwise order of edges embedded around each vertex, for some unspecified plane embedding of the graph. The Hopcroft-Tarjan algorithm can be modified so that when the input graph G is planar, it re-orders the adjacency lists to ensure that G is a planar map [10,11].

Although a given planar map may arise from several distinct embeddings (any face can be the external face), it is convenient sometimes to imagine that an embedding has been given, so we can speak of the faces of the map.

Because we use the adjacency list representation, each edge $\{u, v\}$ possesses entries in the adjacency list for u and that for v: in effect, it is represented by two directed edges (u, v) and (v, u). We assume that adjacency lists are cross-linked, so from the entry for (u, v) one can directly access the inverse edge (v, u) in the other adjacency list. Using these linkages, from any edge (u, v) one can access the edge (v, w) which immediately follows (v, u) in anticlockwise (respectively, clockwise) order in the adjacency list for v.[1]

Suppose that (v, w) follows (v, u) in *clockwise* order around v. There is a unique face to the left of (u, v), and the same face lies to the left of (v, w). By following edges in this way one can trace out a simply cycle of edges on the boundary of the face, with the face to the left of all the edges: if the face is bounded then the boundary is followed anticlockwise, otherwise (with the external face to the left) in clockwise order. See Figure 1.

A planar map G is *triangulated* if for every face F of G, the boundary ∂F is a 'triangle'

[1] This follows the strategy of Guibas and Stolfi [5], who were anticipated in some respects by the mathematician Hamilton [6] in his analysis of the icosian game.

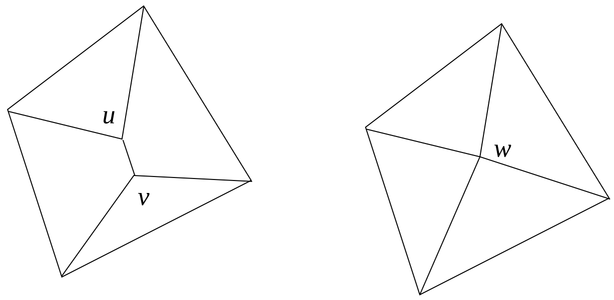

Figure 2: graph contraction.

(a simple 3-cycle). It is possible to triangulate a planar map, by introducing new edges where necessary. (This might seem easy algorithmically, as it is trivial to triangulate a single face; however, unless one is careful one might introduce two copies of the same edge, and the result would not even be a graph. However, G can be triangulated in linear time and space [8,11].)

Given a graph G with edge $e = \{u, v\}$, the *contraction* G/e of G is the graph obtained by replacing the nodes u and v by a single node w, deleting the edge $\{u, v\}$, and replacing all other edges $\{u, x\}$ or $\{v, x\}$ by $\{w, x\}$. If G is a planar graph then G/e is a planar graph; however, if G is a planar map, then G/e is not necessarily a planar map. See Figure 2.

In implementing the algorithm it is more sensible to construct G/e by deleting one of the nodes, v say, and replacing every edge $\{v, z\}$ by $\{u, z\}$ (assuming it is not already an edge of G).

Briefly, the layout algorithm is: given a triangulated planar map G, if G is a triangle then lay it out directly, otherwise choose a suitable (directed) edge e, construct the contracted graph G/e, recursively lay it out, then 'extrude' the edge e so that a layout of G is obtained.

This paper is mostly concerned with a choice of edge e for contraction that will lead to a correct linear-time algorithm.

Disclaimer. The algorithm given here turns out to be very similar to one of Read's, published in 1987 [13]. The algorithms are not identical, however. Our algorithm starts with the notion of contracting an edge; Read's focusses on deleting a vertex. The proofs are quite different.

The author apologises for the somewhat scrappy LEDA code. Actually, the implementation is not linear time; a hash-table is used to access information about graph edges, so the linear time is only average-case. Other than that the implementation runs in linear time.

2 Properties of planar maps.

A well-known property of planar graphs is that they are sparse: a planar graph with $n \geq 3$ nodes has at most $3n-6$ edges and $2n-4$ faces (the figure is exact for triangulated planar maps). One interesting and useful consequence of this is that a planar graph has 'cheap nodes,' in the sense of the following lemma due to Kirkpatrick [9,10]. The degree of a node is the number of edges incident to it.

(2.1) Proposition *For any $k > 0$, a planar graph with n nodes has more than $(k-6)n/k$ nodes of degree less than k.* ∎

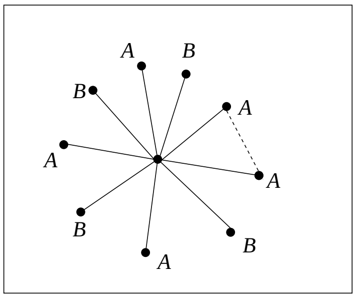

Figure 3: A not an independent set (Lemma 2.2).

We use the phrase *cheap edge* to mean an edge incident to a bounded number of other edges. We have a lemma, similar in spirit to the above proposition, ensuring the existence of at least one 'cheap' edge. In the lemma, if a set X of nodes of G has the property that no two nodes in X are joined by an edge of G, then the set is called 'independent.'

(2.2) Lemma *A triangulated planar map G has at least one edge incident to at most 18 other edges.*

Proof. Let A be the set of nodes in the graph of degree ≤ 10. It is enough to show that this set is not an independent set, so that there is an edge with both ends in A.

Let B the complement of A. Let us assume that B is nonempty (otherwise the result is trivial). For every node u in B there are at least 11 edges incident to u. Let H be the subgraph of G spanned by B: i.e., B forms the set of nodes of H, and the edges of H are those edges of G with both ends in B. H is, of course, planar, so by Proposition 2.1, H has at least one node u of degree ≤ 5. But u has degree ≥ 11 in G. Hence there exist two nodes from A adjacent to u and consecutive in cyclic order around u. (See Figure 3.) If v and w are these two nodes, then $\{v, w\}$ is an edge of G, since G is triangulated. ∎

The above lemma does not apply to non-triangulated graphs: for instance, the tree consisting of just a root with 20 children has no 'cheap edge'.

(2.3) Lemma *The only triangulated planar map with nodes of degree < 3 is the graph with 3 nodes and edges, the '3-cycle.'*

Proof. The graph has no nodes of degree 1 or 0, since if it had, it would have a face F whose boundary included either an edge both sides of which were incident to F, or an isolated point: in either case the boundary would not be a 3-cycle.

Let $\{u, v\}, \{v, w\}$ be edges where v has degree 2. The face to the left of (u, v) is the face to the left of (v, w), so the 3-cycle uvw bounds a face on its left; similarly, it bounds a face on its right. This is only possible if one of the faces is the external face, the outside of the 3-cycle, and the other is its inside; so there are 3 nodes and edges. ∎

We need to strengthen Lemma 2.2 slightly. We suppose that a graph G is given with 3 of its nodes 'marked.' We want a cheap edge avoiding these nodes. Call an edge marked if either of its endpoints is.

(2.4) Lemma *Every triangulated planar map with at least 6 nodes, 3 of them marked, has an unmarked edge.*

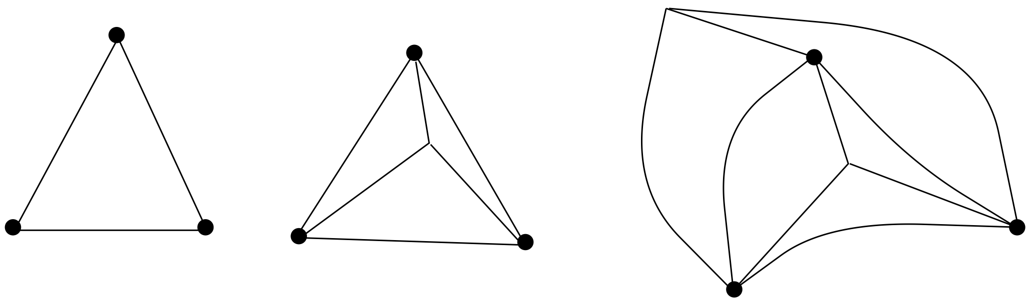

Figure 4: The three possible triangulated maps with no unmarked edges.

PROOF. (The proof will also reveal the only triangulated planar maps with all edges marked: they can have 3, 4, or 5 nodes. See Figure 4.)

Let G be a triangulated planar map with 3 marked nodes u, v, and w, and no unmarked edges. It follows that every face of G is incident to at least two of the nodes u, v, or w, since otherwise at least one of its three boundary edges would be unmarked.

If G has 3 nodes then it is the triangle discussed in Lemma 2.3. Otherwise, every node is incident to at least 3 faces and 3 edges. Claim that $\{u, v\}$, v, w, and $\{u, w\}$ are all edges of G. For, consider any (triangular) face F incident to u: it has at least 2 marked nodes incident, so, without loss of generality, $\{u, v\}$ is an edge of G. Similarly, $\{u, w\}$ or $\{v, w\}$ is an edge of G. Suppose that $\{v, w\}$ is an edge of G, but $\{u, w\}$ is not. Then, since there are at least 3 edges incident to u, there exist two cyclically adjacent edges $\{u, x\}$ and $\{u, y\}$ where x and y are distinct from u, v, and w: hence both x and y would be unmarked and adjacent, since they bound the same face of G. This contradiction shows that $\{u, w\}$ is also an edge of G.

It remains to show that there is at most one node of G inside the triangle uvw, and one outside.

Suppose that there exists a node y inside uvw. It is incident to at least three edges, each with a marked node on it (and y is not marked); hence y is adjacent to u, v, w, and no other node. By standard planarity arguments [10], G cannot have two nodes both inside uvw and both adjacent to u, v, and w. Thus there exists at most one node y inside u, v, w, and it is adjacent to all three marked nodes; similarly, there exists at most one node outside the triangle, and it is adjacent to all three marked nodes. ∎

(2.5) Lemma *A triangulated planar map G, with 3 marked nodes, and at least 6 nodes overall, has an unmarked edge which is adjacent to at most 30 other edges.*

PROOF. Let A be the set of nodes of degree at most 16, B its complement. If B is nonempty, let H be the subgraph spanned by B as in Lemma 2.2, and consider a node u of degree 5 or less in H. It is adjacent to at least 17 nodes in G, of which at most 8 are either in B or in A but marked. Therefore there are at least two unmarked nodes in A, consecutive in cyclic order around u, and there is an edge connecting them adjacent to at most 30 other edges.

If B is empty, then all edges of the graph are incident to at most 30 other edges. By Lemma 2.4, since $n \geq 6$, G contains an unmarked edge. ∎

Recall the definition of a contracted graph G/e. We want a condition such that if G is a triangulated planar map with a particular embedding assumed, then G/e is a triangulated

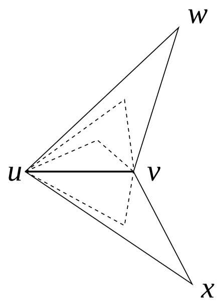

Figure 5: illustrating a non-admissible edge.

planar map arising from a related embedding. Of course, the number n of nodes in G must be ≥ 4: hence G has exactly $3n-6$ edges and $2n-4$ faces. Suppose $n \geq 4$. Let $e = \{u, v\}$, and let x, y be the third nodes on the face boundaries on either side of e. A single edge $\{u, x\}$ of G/e replaces two edges of G; similarly for $\{u, y\}$. The edge $\{u, v\}$ is deleted. Thus G/e has at least 3 fewer edges, so in order that it be a triangulated map, x and y must be the only nodes adjacent to both u and v. See Figure 5. Bearing this in mind,

(2.6) Definition *Given a planar map G with at least 4 nodes, of which 3 are 'marked', an edge e is* admissible *if there are exactly two other nodes x, y adjacent to both ends of e, and at most one endpoint of e is marked.*

The reason for 'marking nodes' is that our algorithm first selects a face to be the external face, marks the nodes on that face as 'external,' and fixes the coordinates of those nodes. Thereafter all new faces will be 'internal.' Since the external nodes are fixed from the start, none of the three external edges can be contracted. However, if $\{u, v\}$ is an edge where u but not v is not external, then the edge can be contracted: v will be mapped to u in $G/\{u, v\}$.

(2.7) Lemma *A triangulated planar map G with at least 4 nodes, the three external nodes being marked, contains a 'cheap' admissible edge, i.e., an admissible edge incident to at most 30 other edges.*

Proof. Let n be the number of nodes in G. The proof is by induction on n. If $n = 4$ or $n = 5$ then the result is clear from Figure 4.

So we assume that $n \geq 6$ and the result holds for all $n' < n$. By Lemma 2.5, G contains an unmarked edge $\{u, v\}$ incident to at most 30 other edges, which is not necessarily admissible: there could be more than two vertices adjacent both to u and to v.

If $\{u, v\}$ isn't admissible then choose a vertex x adjacent both to u and v with the property that there exists no other node x' adjacent to u and v such that x is inside the triangle uvx'. See Figure 6.

Let H be the subgraph inside the triangle uvx (including the sides of the triangle) mark the nodes u, v, and x 'external,' and consider the triangulated planar map H. H must be different from G, because the nodes u and v are not on the external face. Moreover, H contains at least 4 nodes, since G has nodes within the triangle uvx.

Hence, by induction, H contains a cheap admissible edge $\{u', v'\}$. At least one of its endpoints, v', say, must be inside the triangle uvx, and hence all nodes adjacent to v' in G

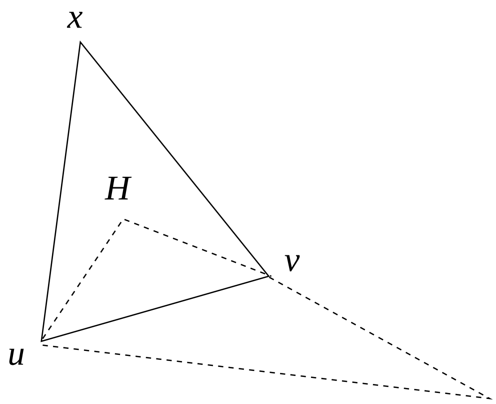

Figure 6: illustrating Lemma 2.7.

must be adjacent to v' in H. Therefore there are exactly two nodes of G adjacent both to u' and v'. Since H contains at most one node which is external in G (namely, x), u' and v' are not both marked in G. Hence $\{u'v'\}$ is an admissible edge in G. ∎

We have seen that given a triangulated planar map G and edge e, if e is not admissible then G/e is not a triangulated planar map. The converse holds as well:

(2.8) Lemma *If G is a triangulated planar map (with at least 4 nodes) and e is admissible, then G/e is a triangulated planar map.*

Proof. We assume a fixed embedding of G, and do not distinguish nodes from vertices or edges from arcs. Let $e = \{u, v\}$, and let x and y the other vertices on the boundary of the two faces incident to e. Delete the edges $\{x, v\}$ and $\{y, v\}$. For each other edge except e incident to v, connecting another vertex w to v, say, replace it by an edge connecting w to u, by displacing it slightly near v and extending it by an arc running close to e and ending at u. With sufficient care the modified edges intersect only at endpoints.[2]

The resulting embedded graph is a planar embedding of G/e. ∎

3 The graph layout algorithm.

The algorithm is simple and recursive: Given as input a triangulated planar map G,

1. First choose which face is the external face. This means traversing one of the face boundaries and marking its incident nodes 'external.' The three nodes on the external face can be embedded directly as the corners of a suitable triangle.

2. If there are only three nodes, we are done. Otherwise traverse the graph, placing all cheap admissible edges on a list.[3] The algorithm continues recursively:

3. Choose a cheap admissible edge e and modify the graph to represent G/e. The list of admissible edges may need to be changed here.

[2]Note that if e were not admissible then this process would introduce multiple edges.

[3]To check if a cheap edge $\{u, v\}$ is admissible, mark all the nodes adjacent to u, then count all the marked nodes adjacent to v: there should be exactly 2.

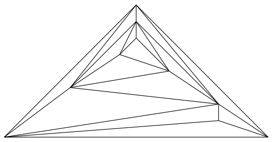

Figure 7: 10-node graph embedded by LEDA routine.

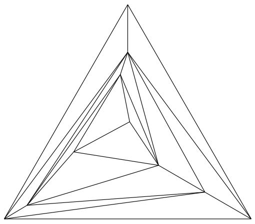

Figure 8: 10-node graph embedded by our algorithm.

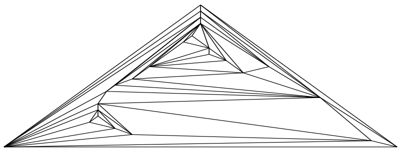

Figure 9: 25-node graph embedded by LEDA routine.

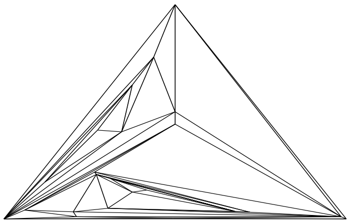

Figure 10: 25-node graph embedded by our algorithm.

4. Recursively, lay out G/e.

5. The edge $e = \{u, v\}$ is mapped to a point u. 'Extrude' this point to a straight line-segment, so that the vertices incident to v are all visible from v. This yields an embedding of G.

With suitable data-structures, the processing in each recursive step is bounded. This algorithm has been implemented using the LEDA C++ library. The implementation 'almost' achieves linear run-time, but not quite: we found it convenient to use a hash-table indexed by graph edges to store bookkeeping information. Figures 7 to 10 show sample layouts of 10-node and 25-node graphs, using an embedding algorithm supplied with LEDA, and the algorithm described in this paper.

4 Acknowledgement

The author is grateful to Christian Uhrig for his considerable help in using LEDA. He is grateful also to Giuseppe di Battista and Peter Eades for information about existing algorithms.

5 References.

1. G. Di Battista, P. Eades, H. de Freysseix, P. Rosenstiehl, and R. Tamassia (eds) (1993). *Graph Drawing '93: proceedings of the ALCOM international workshop on Graph Drawing*, Sèvres, France.

2. G. Di Battista, R. Tamassia, I.G. Tollis (1990). Area requirement and symmetry display of planar upward drawings. ALCOM report 90-77, Università di Roma 'La Sapienza.'

3. P. Bertolazzi, G. Di Battista, G. Liotta, and C. Mannino (1994). Upward drawings of triconnected graphs. *Algorithmica 12:6*,476–497.

4. M. Formann, T. Hagerup, J. Haralambides, M. Kaufmann, F.T. Leighton, A. Simvonis, E. Welzl, G. Woeginger (1990). Drawing graphs in the plane with high resolution. Report B 90-10, Mathematik Serie B, Freie Universität Berlin.

5. L. Guibas and J. Stolfi (1985). Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Trans. Graphics* **4**, 74–123.

6. T.L. Hankins (1980). *Sir William Rowan Hamilton.* Johns Hopkins University Press.

7. J. Hopcroft and R.E. Tarjan (1974). Efficient planarity testing. *JACM* **21**, 549–568.

8. G. Kant (1993). *Algorithms for drawing planar graphs.* Ph.D. dissertation, Computer Science Department, University of Utrecht.

9. D.G. Kirkpatrick (1983). Optimal search in planar subdivisions. *SIAM J. Computing* **12**, 28–35.

10. K. Mehlhorn (1984). *Data structures and algorithms 2: graph algorithms and NP-completeness.* Springer-Verlag.

11. K. Mehlhorn, P. Mutel, and St. Näher (1993). An implementation of the Hopcroft and Tarjan planarity test and embedding algorithm. Technical report MPI-I-93-151, Max-Planck-Institut für Informatik, Saarbrücken.

12. S. Naher (1993). LEDA Manual, Version 3.0. Technical report MPI-I-93-109, Max-Planck-Institut für Informatik, Saarbrücken.

13. R. Read (1987). A new method for drawing a planar graph given the cyclic order of the edges at each vertex. *Congressus Numerantium* **56**, 33-44.

6 LEDA program listing

```
#include <LEDA/array.h>
#include <LEDA/h_array.h>
#include <LEDA/list.h>
#include <LEDA/planar_map.h>
#include <LEDA/plane.h>

        /*************************************
        |        Planar layout algorithm.
        |        Colm O Dunlaing, October 1994.
        |        Works with parametrised planar map M<nnotes*,int>.
        |        The information is...........
        **************************************/

typedef struct {int index; bool external, marked;} nnotes;

        /*************************************
        |     The edge information, ‘nman’ (number of mutually adjacent
        |     nodes) for edges with ‘low’ degree (both ends
        |     having degree <= 16), is the number of nodes simultaneously
        |     adjacent to both ends.  Negative value indicates ’uncomputed.’
        |     The field alpos access place on an ‘admission list.’
        *************************************/

typedef struct {int nman; list_item alpos;} enotes;

void extrude ( list<int> polygon, int vindex, array<double> &x,
                        array<double> &y )
{
        /************************************
        |         polygon gives a list of indices defining
        |         (through arrays x and y) a sequence of points
        |         known to form a simple polygon.  It is also
        |         known that the last list element gives a point
        |         from which all others are visible.
        |         The list presents the polygon in CLOCKWISE order,
        |         so it is to the right of the edge joining
        |         the first pair of corners.  The routine
        |         computes a point inside the polygon from which all
        |         its vertices are visible and stores this in
        |         x[vindex], y[vindex].
        *************************************/
{//DEBUGGING
cerr << "extrude: polygon";
```

```
int t;
forall(t, polygon) cerr << ' ' << t;
cerr << '\n';
}

        int i = polygon.Pop(),                // `principal' corner.
            j = polygon.Pop(),                // next, anticlockwise.
            k = polygon.head();                // next, clockwise.

        point p(x[i],y[i]);
        point q;

        if ((y[i]-y[j])*(x[k]-x[i])+(x[j]-x[i])*(y[k]-y[i])>0)
{cerr << "angle jik is acute\n"; // DEBUGGING
                q=point((x[j]+x[k])/2, (y[j]+y[k])/2);
}//DEBUGGING
        else
{cerr << "angle jik is reflex\n";//DEBUGGING
                q=point(2*x[i]-(x[j]+x[k])/2, 2*y[i]-(y[j]+y[k])/2);
}//DEBUGGING

        line L(p,q);
cerr << "p= " << p << "; q= " << q << '\n';

        /*************************************
        |        The line L, directed from p, points
        |        into the interior of the polygon.  The
        |        iteration below recomputes q as the
        |        intersection point of a side of the polygon
        |        with L, closest to p.
        *************************************/

        bool started = false;
        while (!polygon.empty())
        {
                k=polygon.Pop();
cerr << "Intersection r of Lp,q with L" << j << "," << k << "(L";
cerr << point(x[j],y[j]) << ", L" << point(x[k],y[k]) << ")\n";
                point r;
                if (L.intersection(line(point(x[j],y[j]),point(x[k],y[k])),r))
                {
cerr << "...is " << r << '\n';
                double d=
                        (q.xcoord()-p.xcoord())*(r.xcoord()-p.xcoord())
                                +
```

```
                        (q.ycoord()-p.ycoord())*(r.ycoord()-p.ycoord());
                double e=
                        (q.xcoord()-p.xcoord())*(q.xcoord()-p.xcoord())
                                +
                        (q.ycoord()-p.ycoord())*(q.ycoord()-p.ycoord());
cerr << "d = "<<d<<", e = "<<e<<"; q ";
                if (d>0 && ((!started) || d<e ))
                        q=r;
                else//DEBUGGING
        cerr << "NOT ";//DEBUGGING
cerr << "replaced by r\n";
                started = true;
                }
                j=k;
        }
        x[vindex] = (p.xcoord()+q.xcoord())/2;
        y[vindex] = (p.ycoord()+q.ycoord())/2;
}

void insert_a_l
   ( PLANAR_MAP<nnotes*, int> &M, edge e, h_array<edge,enotes*> &note ,
                        list<edge> & admission )
{
        edge f = M.reverse(e);
        if (note[e]->alpos != nil ||
                 note[f]->alpos != nil )
        return;
        note[e]->alpos = admission.append(e);
//cerr << "insert_a_l: size now " << admission.size() << "\n";
}

void delete_a_l
   ( PLANAR_MAP<nnotes*, int> &M, edge e, h_array<edge,enotes*> &note,
                        list<edge> & admission )
{
        edge f = M.reverse(e);
        list_item pos = note[e]->alpos;
        if (pos!=nil)
                admission.del_item ( pos );
        pos = note[f]->alpos;
        if (pos!=nil)
                admission.del_item ( pos );
        note[e]->alpos = note[f]->alpos = nil;
//cerr << "delete_a_l: size now " << admission.size() << "\n";
}
```

```
void compute_nman ( PLANAR_MAP<nnotes*, int> &M,
                        node u, h_array<edge, enotes*> &note ,
                        list<edge> &admission )
{
        /***********************
        |    Computes the nman information
        |    for every edge incident to u
        |    both of whose ends have degree <= 16.
        ***********************/
//cerr << "compute_nman\n";
        if (M.indeg (u) > 16)
                return;
        node v;
        forall_adj_nodes ( v, u )
                M.inf(v)->marked = true;
        edge e;
        forall_adj_edges ( e, u )
        {
          v = M.target (e);
          int count;
          if (M.indeg(v)>16 || (M.inf(u)->external && M.inf(v)->external))
                  count = 3; // dummy value, forces edge to be
                             // deleted from admission list
          else
          {
                  count = 0;
                  node w;
                  forall_adj_nodes ( w, v)
                    if (M.inf(w)->marked)
                      ++count;
                  note[e]->nman = count;
                  note[M.reverse(e)]->nman = count;

          if ( count==2 )
          {
            if (!M.inf(v)->external)
              insert_a_l (M, e, note, admission);
            else
              insert_a_l (M, M.reverse(e), note,
                admission);
          } else
          {
            delete_a_l ( M, e, note, admission );
          }
        }
```

```
        forall_adj_nodes ( v, u )
                M.inf(v)->marked = false;
        }
}



void layout ( PLANAR_MAP<nnotes*, int> &M,
                list<edge> &admission,
                h_array<edge, enotes*> &note,
                array<double> &xpos,
                array<double> &ypos )
{
cerr << "Entering layout: " << M.number_of_nodes() << " nodes; "
        << M.number_of_edges() << " edges;  "
        << admission.size() << " on a.l.\n";

        /************************
        |        if 3 nodes, return.
        ************************/

        if (M.number_of_nodes()==3) return;

        /************************
        |        if not 3 nodes and empty admission list, error.
        ************************/

        if (admission.empty())
        {
                error_handler(1, "abort - empty admission list");
        }

        /************************
        |        admissible edge (u,v) chosen.
        |        v will be deleted, the map recursively laid out,
        |        and coordinates calculated for v.
        |        Record the index of the node v.
        ************************/

        edge uv = admission.head();
        delete_a_l ( M, uv, note, admission );
        node u=M.source(uv), v=M.target(uv);
        int vindex=M.inf(v)->index;
//cerr << "u index " << M.inf(u)->index << " v index "
       << M.inf(v)->index << '\n';
```

```
/*******************************
|        Form a list of all edges from v, in
|        clockwise order, ending with the reversal
|        vu of uv.
*******************************/

list<edge> v_adj;
edge vu = M.reverse(uv);
edge e = vu;
do
{
        v_adj.push ( e );
        e = M.cyclic_adj_succ ( e );
} while ( e != vu );

/*******************************
|        Delete all edges incident to v, replacing
|        all but three of them by
|        edges incident to u.  It is convenient
|        to do this by successive deletion and
|        insertion of edges.
|        Store the indices of the endpoints of the deleted
|        edges as a list 'polygon.'  Effectively, this
|        describes the boundary of the face remaining when
|        v is deleted.  The polygon is listed ANTICLOCWISE.
*******************************/

list<int> polygon;

e = M.cyclic_adj_succ ( uv );

while ( ! v_adj.empty() )
{
        edge f = v_adj.pop();
        polygon.append ( M.inf(M.target(f))-> index);
        delete_a_l ( M, f, note, admission );
        note[f]=note[M.reverse(f)]=nil;
        M.del_edge (f);
        if ( v_adj.size() >= 3 )
        {
                edge g = M.reverse ( v_adj.head() );
                e=M.new_edge(e,g);
                enotes *enote = new enotes;
                enote->alpos = nil; enote->nman = -1;
```

```
                note[e] = enote;
                enote = new enotes;
                enote->alpos = nil; enote->nman = -1;
                note[M.reverse(e)] = enote;
        }
    }

    M.del_node(v);

    /*************************************
    |       Node v is now deleted. (But vindex records its index.)
    |       Compute the nman field for all
    |       edges with at least one end adjacent
    |       to u.  Note: compute_nman returns immediately
    |       if the node involved has degree > 16.  This
    |       could cause difficulty by not deleting from
    |       the admission list edges whose node degrees
    |       had increased.  However, only one node can have
    |        increased degree, namely, u; and all edges with
    |       one end incident to u and the other end of low
    |       degree will be processed by compute_nman.
    ************************************/

    forall_adj_nodes ( v, u )
            compute_nman ( M, v, note, admission );

//DEBUGGING: make a copy of M; it will be output upon return from recursion

    PLANAR_MAP<nnotes*,int> tempM( M );

    /****************************
    |       Recursively layout M.
    ****************************/

    layout ( M, admission, note, xpos, ypos );


    /********************************
    |       'extrude' the edge {u,v}.  This means taking the
    |       list of node indices for all nodes around v, and storing
    |       in the array location for v (i.e., vindex)
    |       the coordinates of a point from
    |       which all nodes around v are visible.
    ********************************/
```

```
        extrude ( polygon, vindex, xpos, ypos );

}


main(){
        GRAPH<nnotes*,int> G;
        int n,m,i,j,k;
        node u;
        cin >> n >> m;
// n nodes and m edges
        if ( n < 3 || m!= 6*n-12)
                exit(-1);
          h_array<edge,enotes*> note;
        array<node> Gnode(0,n-1);
//cerr << "main: creating nodes\n";
        for (i=0; i<n; ++i)
        {
                u=G.new_node();
                nnotes *nt = new nnotes;
                nt->index = i; nt->marked = nt->external = false;
                G.assign ( u, nt );
                Gnode[i]=u;
        }

        edge e;

        for (k=0; k<m ; ++k)
        { cin >> i >> j;
                  e= G.new_edge( Gnode[i], Gnode[j] );
        }

//cerr << "main: G now exists. Create M with edge note bindings.\n";

  PLANAR_MAP<nnotes*, int> M(G);

        forall_edges ( e, M )
        {
                enotes *ent = new enotes;
                ent->nman = -1; ent->alpos = nil;
                  note[e] = ent;
        }

//cerr << "main: edges now exist for planar map M.\n";
  list<edge> admission;
```

```
    array<double> x(0,n-1);
    array<double> y(0,n-1);
    e = M.choose_edge();
        // face to the right of e will be chosen as external.
    i = M.inf ( M.source(e) ) -> index;
    M.inf (M.source(e)) -> external = true;
    j = M.inf ( M.target(e) ) -> index;
    M.inf (M.target(e)) -> external = true;
    e = M.cyclic_adj_succ(M.reverse(e));
    k = M.inf ( M.target(e) ) -> index;
    M.inf (M.target(e)) -> external = true;
    x[i] = 0; y[i] = 0;
    x[j] = 10; y[j] = 0;
    x[k] = 5;  y[k] = 8.66025403765;
        // external face nodes form equilateral triangle of
        // side 10.
  forall_nodes (u, M)
    if (M.indeg(u)<=16)
      compute_nman ( M, u, note, admission );

    cout << n << '\n'; // First output datum is overall
                       //  size of graph; last is -1.

    layout ( M, admission, note, x, y );

    cout << n << ' ' << m << '\n';
    for ( i=0; i<n; ++i )
        cout << i << ' ' << x[i] << ' ' << y[i] <<'\n';
    for ( i = 0; i<n; ++i )
        {
          u = Gnode[i];
          node v;
          forall_adj_nodes ( v, u )
                cout << i << ' ' << G.inf ( v ) -> index << '\n';
        }
    cout << "-1\nEnd of data indicated by nonpositive node count.\n";
}
```